ASLan++ — A formal security specification language for distributed systems

David von Oheimb[1] and Sebastian Mödersheim[2]

[1] Siemens Corporate Technology, IT Security, Munich, Germany,
David.von.Oheimb@siemens.com, ddvo.net
[2] DTU Informatics, Technical University of Denmark, Lyngby, Denmark
samo@imm.dtu.dk, imm.dtu.dk/~samo

Abstract. This paper introduces ASLan++, the AVANTSSAR Specification Language. ASLan++ has been designed for formally specifying dynamically composed security-sensitive web services and service-oriented architectures, their associated security policies, as well as their security properties, at both communication and application level.
We introduce the main concepts of ASLan++ at a small but very instructive running example, abstracted form a company intranet scenario, that features non-linear and inter-dependent workflows, communication security at different abstraction levels including an explicit credentials-based authentication mechanism, dynamic access control policies, and the related security goals. This demonstrates the flexibility and expressiveness of the language, and that the resulting models are logically adequate, while on the other hand they are clear to read and feasible to construct for system designers who are not experts in formal methods.

Keywords: services, security, specification language, formal analysis

1 Introduction

Formal Security Analysis. Security in distributed systems such as web services and SOA is very difficult to achieve, because often the security problems are very subtle. Even systems that are simple to describe (such as the famous three-line Needham-Schroeder Public Key protocol) may have weaknesses that go unnoticed for years even when the system has been carefully studied [9]. Formal specification and verification of such systems can help to uncover weaknesses before they can be actually exploited. Especially automated verification tools can help to find the needle in the haystack — one trace of the system that violates the security goals among an enormous number of traces that are fine.

Over the last decade, formal verification for security has made a lot of progress. In the late 90s, automated protocol verification tools began to emerge that focussed on simple security protocols that can be described by an exchange of messages (e.g., in Alice&Bob-style notation). Despite being small systems, their verification is very challenging, in particular considering that an intruder

has an unbounded choice in constructing messages, which may involve algebraic properties of the cryptographic primitives. Moreover one cannot give a bound on the number of sessions that can be executed in parallel. These problems are now well understood, both theoretically in terms of complexity and decidability [21,12,15], and in terms of methods and tools that are practically feasible automated verification [8,1,13,16].

Limitations of Security Protocol Analysis. The focus of simple security protocols is however quite limited, ignoring a lot of aspects that play a crucial role in distributed systems and that often *are* relevant for security.

The first very common aspect that falls out of the simple structure of security protocols is non-linear communication. For instance, a (web-) server typically listens for requests that must be in one of several types of formats; depending on the request, the server will start an appropriate workflow, possibly contacting other servers that implement subtasks of the workflow, and then finally give a response to client who sent the initial request.

This brings us immediately to a second aspect: the described transaction may sometimes *not* be independent from all other transactions, but for instance may be related via dynamic distributed state. For instance, in case of an online shop, a database maintained by the server may contain the set of all processed orders and their status, the set of all registered customers, and other related information. Processing different requests may depend on this database, for instance a registered user can send a request to see all her recent orders — provided the user can authenticate herself by username and password or maybe by a cookie. Another subsequent request could then be to cancel or change an order that has not yet been shipped. These aspects are completely outside the realm of simple security protocols where different sessions are essentially independent and the only information shared between different sessions are static long-term keys.

A third important aspect concerns the relation to dynamic security policies. For example, when a server receives a request from a client to access a resource it controls, it may need to check whether the particular client has the necessary access rights. These access rights may not be static but may for instance depend on who is a member of the group that owns a particular resource, and these memberships may change over the time. The change of group memberships may itself be transactions of the system that is again governed by some access control policies, e.g., only members of a certain role, say *manager*, are authorized to change group memberships.

AVANTSSAR and its Specification Language. The EU-funded Project AVANTSSAR has been concerned with developing a formal specification language and automated verification methods and tools to handle systems at design level in which all these three aspects are relevant: non-linear work-flow, relationships between workflows (for instance via databases), and access control policies. In this paper, we describe the AVANTSSAR Specification Language

ASLan++ [4], which has been developed as a joint effort by the partners of the project. The design goal of ASLan++ were

1. expressiveness sufficient to describe the security-relevant aspects of service-oriented architectures as described above,
2. ease of use for systems designers, in particular being close to the way designers think about and describe such systems, allowing to abstract from details whenever they are not relevant or can be "factored out",
3. compatibility with existing and emerging verification methods so that automatically analyzing specifications is feasible at least for small number of parallel processes, without being biased to a particular method.

Structure of this Paper. In this paper, we discuss the main concepts of ASLan++ and how they can be used for modeling the most relevant aspects of service-oriented architectures. We also briefly discuss the rationale behind some design decisions and some consequences for the verification methods, drawing in particular from our experience in modeling a number of larger case studies.

For introducing ASLan++, we employ a small but very instructive running example specification. Its full text may be found in the appendix, while in the following sections we describe it piece-by-piece, progressing from basic structuring of the specification over its procedural aspects to security policies, communication properties, and security goals.

The example describes part of a company intranet scenario. Employees may access files according to a dynamic access control policy. A central server keeps track of the access rights. Both managers and employees can influence the policy.

2 Specification Structure and Execution

2.1 Specifications

An ASLan++ specification of a system and its security goals consists of a hierarchy of *entities*. An entity may import other entities contained in separate files, which in turn contain a hierarchy of entity declarations. The top-level entity, usually called `Environment`, serves as the global root of the system being specified, similarly to the "main" procedure of a program. In our example specification, the `Environment` has two sub-entities: `Session` and `Employee`, where the former has in turn two sub-entities: `Server` and `Manager`. The `Manager` entity, for example, is used to describe the behavior of any honest manager as well as the security requirements that can be stated from her perspective.

2.2 Entities and agents

Entities are the major ASLan++ building blocks, which are similar to classes in Java or roles in HLPSL [10] and other security protocol specification languages. Entities are a collection of declarations and behavior descriptions. They can have parameters and local variables, with the usual nested scoping w.r.t. sub-entities.

Entities are like blueprints that can be instantiated to any number of processes (or threads), each executing the *body* of the entity. With the exception of sets, the parameters of an entity have call-by-value semantics: the entity obtains a copy of the values and may change them without side-effects for the "calling" process. On the other hand, one can easily model shared data [4, §2.4].

Each entity has an explicit or implicit formal parameter `Actor`, which is similar to `this` or `self` in object-oriented programming languages. The value of `Actor` is the name of the agent playing the role defined by the entity. This is important for defining the security properties of the entity.

The entity and instance structure of our example is as follows, where entity and variable names are uppercase, while constant and type names are lower-case. ASLan++ comments start with a "`%`" symbol and extend until end of the line.

```
entity Environment {
  ...
  entity Session (M, S: agent) {

    entity Server(M, Actor: agent) {
      ...
    }
    entity Manager(Actor, S: agent) {
      ...
    }
    body { % of Session
      ...
      new Server (M,S);
      new Manager(M,S);
    }
  }
  entity Employee(Actor, S: agent) {
    ...
  }
  body { % of Environment
    ...
    any M. Session(M,centralServer);
    new  Employee(e1,centralServer);
    new  Employee(e2,centralServer);
  }
}
```

There is (at least) one instance of entity `Session`, each invoked by a statement `any M. Session(M,centralServer)` (described later). It has two formal parameters (of type `agent`): `M` refers to the agent playing the manager role, while `S` holds the name of the server. Each session launches in parallel a thread of the `Server` and a `Manager` instance, by statements like `new Manager (M,S)`. The session(s) runs in parallel with two (or more) `Employee` instances.

The `Manager` entity has two parameters: `Actor` is used to refer to herself, while `S` holds the name of the server she is going to interact with. The parameters of the `Employee` entity are analogous to `Manager`.

Instances of the `Server` entity will actually obtain the name of the manager via the manager's messages described below. Still, for the sake of relating entities for the security goals, we need to give `M`, the variable that will hold the manager's agent name, as a formal parameter of `Server`. The other parameter of `Server` is, as usual, the `Actor`.

Note that while each instance of `Manager` and `Employee` (typically) has a different agent playing the respective role, for example referred to by the constants `e1` and `e2` used for employees, there is just a single constant `centralServer` used as actor of the `Server` entity. This is how we model that the server is global.

2.3 Execution Model

The instantiation of an entity is *in parallel*: the caller starts a new process that runs in parallel to the caller. A subtle point is the granularity at which parallel processes can be interleaved. Consider that a web server may make quite a number of intermediate computations between receiving a request and sending a reply. Running in parallel with other processes (e.g., other instances of the same server that currently serve a different request) produces an exponential number of interleavings, which is difficult to handle for many verification methods. There is also a number of classical problems attached, e.g., if we think of a two threads of the server checking and modifying the database, this can easily lead to race conditions. For ASLan++ we have chosen a particular way to deal with interleavings. Whenever an entity receives a message and then acts upon that, we consider its subsequent activity *atomic* up to the point where the entity goes back into a state of waiting for further messages. The reason is quite pragmatic: we get a coarse interleaving model that is feasible for verification tools without the user having to code tool-related optimizations into the specification (i.e., declaring atomicity-blocks to help the tools). At the same time, this can be regarded as a reasonable model for many situations: when the server's computation is related to a shared resource, e.g., reading from a database and then writing a change into the database, it is clear that in the implementation that process should get a lock on the server so that other processes do not change the database in between. ASLan++ thus allows to abstract from such locking mechanisms, and in fact they are often not the focus of a security verification. However, if desired, ASLan++ also allows to declare "custom breakpoints" (overriding the default atomicity behavior) to model a finer interleaving model.

ASLan++ offers experimental support for constraints on the global system run via LTL formulas, which may be used to specify e.g., fairness assumptions.

2.4 Dishonest agents and the intruder

The attacker is known as the *intruder* and can be referred to by the constant `i` (of type `agent`). Yet we allow the intruder to have more than one "real name".[3] To this end, we use the predicate `dishonest` that holds true of `i` and of every pseudonym (i.e., alias name) `A` of `i`.

[3] The intruder may have several names that he controls. This reflects a large number of situations, like an honest agent who has been compromised and whose long-term keys have been learned by the intruder, or when there are several dishonest agents who collaborate. This worst case of a collaboration of all dishonest agents may be simply modeled by one intruder who acts under different identities.

As long as the actual value of the `Actor` parameter of an entity is an honest agent, the agent faithfully plays the role defined by the entity. If the `Actor` parameter value is dishonest already on instantiation of the entity, which is typically the case for some of the possibilities included in symbolic sessions (cf. 2.6), the body of the entity is ignored because the intruder behavior subsumes all honest and dishonest behavior.

We also allow that an entity instance gets compromised later, that is, the hitherto honest agent denoted by the `Actor` of the entity becomes dishonest. Once an agent has become dishonest, for instance because it has been corrupted, it can never become honest again.

2.5 Declarations

An entity may contain declarations of types, variables, constants, functions, macros, (Horn) clauses, and algebraic equations.

Unless declared non-public, constants and functions are public, such that the intruder knows them and thus may (ab-)use them. Moreover, function symbols are by default interpreted in the free term algebra (modulo algebraic equations), such that they are by default invertible in each argument. This conveniently reflects the typical behavior of message constructors, like the ones declared in our example:

```
login (agent,symmetric_key): message;
changeGroup (agent,agent set,agent set): message;
assignDeputy(agent): message;
requestAccess(file): message;
grantedAccess(file): message;
 deniedAccess(file): message;
```

where the types in parentheses specify their argument types.

Message constructors abstract from the actual implementation details of how messages are actually encoded. Essentially the only property we rely on is their invertibility, such that e.g., the intruder may obtain `A`, `G1`, and `G2` from knowing `changeGroup(A,G1,G2)`. Since often a function application term is better readable when the first argument is written before the function symbol, ASLan++ offers syntactic sugar for this, such that we can equivalently write in "object-oriented style": `A->changeGroup(G1,G2)`. The message constructors just mentioned, as well as the remaining symbols declared in the global `symbols` section, will be described in more detail below where appropriate.

Types may have subtypes, e.g., the (built-in) relation `agent < message` means that any value of type `agent` may be used in a context where a value of type `message` is expected. The type `message` includes all those values that may be sent over the network, in particular concatenation `M1.M2` and tuples `(M1,M2)` of sub-messages `M1` and `M2`. For "atomic" values in messages, one may use the subtype `text`, which may be dealt with more efficiently during model-checking. For instance, we declare an abstract type of files (or better: file identifiers) as

```
types
  file < text;
```

Sets, which are passed by reference, are not a subtype of `message`, such that they cannot be directly sent as messages. [4] Sets and tuples have parameters for their element types, e.g., `nat set` and `agent * message`).

Symbols may also be declared in the respective sections of the various entities, in particular the local variables that their instances use internally. For instance, both `Manager` and `Server` declare

```
symbols
  Cookie: cookie;
```

where `cookie` is a subtype of `text`.

2.6 Statements

Statements may be the usual assignments, branches and loops, but also nondeterministic selections, assertions, generation of fresh values and of new entity instances, transmission of messages (i.e., send and receive operations), and introduction or retraction of facts, which represent state-dependent truth values.

The `select` statement is typically used within the main loop of a server, as it is the case in our example for the `Server` entity:

```
body {
  while(true) {
    select {
      on(... & ... ): {
        ...
      }
      ...
      on(?A *->* Actor: requestAccess(?F)): {
        ...
      }
    }
  }
}
```

Such a statement handles a variety of potential incoming requests or other events such as timeouts. It checks the guards given, blocking as long as no guard is fulfilled, then nondeterministically chooses any one of the fulfilled guards and executes the corresponding statement block. The evaluation of the chosen guard assign variables that are written with the `?` symbol before their name. For instance, the guard

```
on(?A *->* Actor: requestAccess(?F)): { ... }
```

(where in this context the decorated arrow symbol `*->*` denotes a communication channel with certain properties, as we will describe in section 4) can fire when a `requestAccess` message has been received from any authenticated agent `A` for any file `F`. When this guard is chosen, the values of these two variables are set according to the actual values received. Then in response the compound statement enclosed by the brackets `{ ... }` is executed.

Entity generation, introduced by the keyword `new` or `any`, instantiates subentities. This is only allowed for direct sub-entities, such that static and dynamic

[4] In [4, §2.5], we describe several possibilities to communicate sets.

scoping coincide. In our example, the `Session` entity creates new instances of the server and the `Manager` entity:

```
new Server (M,S);
new Manager(M,S);
```

These run in parallel and in this case happen to obtain on creation the same actual parameter values, `M` and `S`.

Symbolic entity generations, introduced by `any`, are a convenient shorthand for loosely instantiating an entity, in the following sense: the bound parameters of the entity, as indicated by the given list of variables, allows to explore all possible values, from the domain of their type (which may be any subtype of `message`). An optional guard, which may refer to the variables listed, constrains the selection. This mechanism is typically used to produce so-called *symbolic sessions*, where the bound variables range over type `agent`, such that (unless further constraints exist) their values include `i`, the name of the intruder.

In our example, we symbolically instantiate the `Session` entity by

```
any M. Session(M, centralServer);
```

Note that since we did not constrain the agent value for `M`, it may be in fact the intruder. The model checkers will use this freedom to look for attacks for both honest and dishonest instantiations for `M`.

2.7 Terms

Terms may contain variables (e.g., `A`), constants (e.g., `e1`), and function applications (to be more precise: function symbols applied to first-order terms, e.g., `requestAccess(F)`) including infix right-associative message concatenation, e.g `M1.M2`) and tupeling (e.g., `(A,b2,0)`). Set literals are written as usual (e.g., `{A,B,C}`), while the basic operator on sets is the `contains` function, where the presence of the fact `Set->contains(X)` means that `X` is a member of `Set`.

3 Policies and Transitions

ASLan++ provides an extremely powerful way to specify security policies and their interaction with the dynamic system defined by the entities given in the specification. For simplicity, let us refer to the latter system in the following simply as *the transition system*. Policies are specified by a set of Horn clauses, e.g., stating that a person can get access to some resource if certain conditions are met. In our running example, there are only two such rules:

```
clauses
  accessDirect(A,G,F): A->canAccess(F) :- G->isOwner(F)  & G->contains(A);
  accessDeputy(A,B,F): A->canAccess(F) :- A->deputyOf(B) & B->canAccess(F);
```

These rules make use of the following user-declared predicate symbols:

```
  canAccess(agent    ,file): fact;
  isOwner  (agent set,file): fact;
  deputyOf (agent    ,agent): fact;
```

3.1 Predicates and Facts

Instead of the usual type *bool* for truth values, ASLan++ uses the type `fact`. Terms denoting atomic propositions, generally knows as *predicates*, are represented by functions with result type `fact`. The question whether an atomic proposition holds or not is expressed by the (non-)existence the respective predicate term in a global "fact space". Facts may be combined with the usual logical operators in LTL formulas to express goals, and they are also used in conditions (in `if`, `while`, and `select` statements) known as *guards*.

By default a fact does not hold, but it may be explicitly introduced (simply by writing it as an ASLan++ statement) and retracted. The constant `true` is introduced automatically, while the constant `false` is never introduced. Facts may also be generated by Horn clauses, as described next.

3.2 Horn clauses

The first above rule says that an agent A can access a file F if A is a member of a group G that is the owner of F. The second rule says that A can access file F if A is a deputy of another agent B who has access to F. Note that it is only for the sake of simplicity of the example that this latter rule models a "complete delegation" of all access rights while most real systems would make more fine-grained delegations.

The symbols A, B, G, F are variables that can be instantiated with arbitrary values and hence are regarded as "parameters" of the rules; this allows in the output of (attack) traces to clearly announce which rule with which values of the parameters had been applied. Note that the second rule is "recursive": if A is the deputy of B and B is the deputy of C, then A also gets access to everything that C has access to — and such a line of deputies can be extended ad libitum, implying delegation of access rights along this line.

It is important to see that these rules are positive formulations of access control conditions: A gets access to a file F if and only if there is *some* way to derive `A->canAccess(F)` with the Horn clauses. We do not allow negative formulations such as "A does *not* get access if ...". This has the advantage that ASLan++ policies can never be *inconsistent* in the sense that one rule allows access while another one would deny it. The price that we pay for this is that it is harder in ASLan++ to formulate a higher-level policy that overrides the judgements of a lower level policies; we discuss this below. Observe that by allowing arbitrary definite first-order logic Horn clauses, this alone gives a Turing-complete programming language (namely a subset of Prolog). [5] This expressivity implies that derivability in ASLan++ policies is in general undecidable. There are several ways to restrict this concept to decidable fragments, e.g., allowing only primitive recursion. It was part of the language design of ASLan++ *not* to commit to such a particular restricted fragment, which may be specific to a verification

[5] There is even some (experimental, so far) support for (in-)equalities as side conditions on the right-hand side of clauses.

method. Our method thereby allows to formulate policies in very different ways, e.g., SecPAL and DKAL policies [6,18] can be specified.

It is crucial to first distinguish two kinds of facts, namely the *state facts*: those explicitly introduced by the transition system, and *policy facts*: those more implicitly "generated" by Horn clauses. In our example, `canAccess` is the only policy fact, because it is the only fact that can be produced by the policy rules. All the other facts are state facts. We will come back why we must insist on this distinction.

3.3 Policy Interaction

There are now two ways how the policies can interact with the transition system that we describe by the ASLan++ entity specifications and their instantiations.

First, transitions of an entity can depend on the judgement of policies. For our example, consider the transaction where an authenticated user requests access to a file: the server governing file access should first check whether the policy actually allows this user access to the requested file. Here is the code snippet of the server's behavior (thus `Actor` is `centralServer` here):

```
on(?A *->* Actor: requestAccess(?F)): {
  if (A->canAccess(F))
    Actor *->* A: grantedAccess(F);
  else
    Actor *->* A:  deniedAccess(F);
}
```

The response of the server, either `grantedAccess(F)` or `deniedAccess(F)`, depends on whether `A->canAccess(F)` holds, which is determined by the two Horn clauses as explained above.

The second way that policies can interact with the transition system is just the other way around: the transition system can generate and retract state facts on which the Horn clauses depend. For instance, there can be transitions that change who is the owner of a file, or who is member of which group or who is deputy of whom, and this has an immediate effect on the access rights via the rules. In our example, let us consider that a manager M can tell the server that a certain employee A changes from a group G_1 to a group G_2, so that the server updates the group membership information. Here is the code snippet from the point of view of the server (i.e., `Actor`):

```
on(M *->* Actor: (?A->changeGroup(?G1,?G2)) & ?G1->contains(?A)): {
  retract(G1->contains(A));
  G2->contains(A);
}
```

Like with the messages sent by an employee, here the manager's command is transmitted on a secure channel (including authentication of the manager), and again the command is abstracted into the message constructor `changeGroup` that has the relevant information (the agent A that changes group, and the source and destination group) as parameters. The server just *retracts* the fact that A is a member of G_1 and *introduces* the fact that G_2 now contains A. Note the command is simply ignored if A is not a member of group G_1 at the time the

command is received; in a more detailed model, one would include a feedback message (whether the command was accepted or not) to the manager.

3.4 Concrete Policy Example

Let us consider the consequences of the transition just described for our policy. For concreteness, let us consider a state where we have a manager m_1, three employees e_1, e_2 and e_3, and two groups $g_1 = \{e_1, e_2\}$ and $g_2 = \{e_3\}$. Consider moreover files f_1, f_2, where group g_i owns file f_i, and that initially there are no deputy relations. All this is formulated by the following contents of the `Environment` declaration:

```
symbols % for the concrete access examples
  m1: agent;
  e1, e2, e3: agent;
  g1, g2: agent set;
  f1, f2: file;
  ...

body { % of Environment
  % for the concrete access examples:
  m1->isManager;
  g1->contains(e1); g1->contains(e2);
  g2->contains(e3);
  g1->isOwner(f1);
  g2->isOwner(f2);
  ...
}
```

By our access control rules, e_1 and e_2 can access f_1 and e_3 can access f_2.

When a manager successfully issues the command `e1->changeGroup(g1,g2)`, this implies that e_1 looses her or his access to f_1 but gains access to f_2. Thus, the access rights are obtained as the *least closure* of the state facts under the policy rules: everything that can be derived from the current state by the policy is true, everything else is false.

To illustrate the effects of state transitions to the policy in more depth, let us consider another transaction where A assigns B as her deputy:

```
on(?A *->* Actor: assignDeputy(?B)): {
  B->deputyOf(A);
}
```

Consider that in this way e_1 becomes deputy of e_2 while both are still in group g_1. If the transfer of e_1 from group g_1 to g_2 is performed in this situation, e_1 gets access to f_2, but it does not loose the access to f_1. This is because access to f_1 is still derivable through the deputy relation: e_1 has access to everything that e_2 has access to (via the second policy rule), and e_2 is still a member of g_1 and thus has direct access to f_1 (via the first policy rule).

This illustrates how expressive the combination of transitions and policies actually is. In particular, there can be several independent reasons why an agent has access to a particular resource. Each of these reasons can change dynamically when people enter or leave groups, become deputies of others or stop being deputies. If one reason for access is removed by a transition, but another reason remains, then also the access right remains. Once all reasons are removed, also

the access right is gone. In the previous example, if e_1 stops being deputy of e_2 (say, because e_2 returns from vacation, which can be modeled by a straightforward `revokeDeputy` command) then with that also the reason for access to f_1 is removed, and since no other reason is left, e_1 no longer has access to f_1.

3.5 Meta Policies

Of course this example has been deliberately kept simple, but let us now review briefly how certain more complex aspects can be modeled. One may model the hierarchical structure in a company and model that one inherits the access rights of one's subordinates:

```
accessSuperior(A,B,F): A->canAccess(F) :- A->superiorOf(B) & B->canAccess(F);
superiorDirect(A,B)  : A->superiorOf(B):- A->managerOf(B);
superiorTrans (A,B,C): A->superiorOf(C):- A->superiorOf(B) & B->superiorOf(C);
```

This shows a different application of the Policy/Horn clauses: mathematically speaking, we define the relation `superiorOf` as the *transitive closure* of the `managerOf` relation. Intuitively, `managerOf` gives the direct superior and is a relation controlled by the transition system just like the other state facts like `deputyOf` etc.; while `superiorOf` yields all superiors over any number of hierarchy levels, and this is "immediately computed" depending on the state of `managerOf`.

This example of "superiors can access everything that their subordinates can access" can be regarded as a *meta policy*, i.e., actually a policy about policies or giving boundaries to policies. This is increasingly important because policies may be expressed (formally) at different levels, e.g., there may be policies at the level of workgroups or divisions of a company, or at the level of the company itself, or on top of that policies required by governmental law.

We have just seen an example of a positive top-level policy, which is easy to integrate. More difficult are negative top-level policies. Take the following negative meta policy as an example: one cannot assign a deputy outside one's own group. This aims at preventing the situation in the above example where e_1 still has access to a file of his old group because he is deputy of an old group member e_2. We cannot directly formulate such negative conditions in the Horn clauses of ASLan++, but we could code it indirectly into the transition for assigning deputies:

```
on(?A *->* Actor: assignDeputy(?B) & ?G->contains(?A) & ?G->contains(?B)): {
   B->deputyOf(A);
}
```

Here the first condition `G->contains(A)` determines *one* group $?G$ that A is member of — in fact we have not explicitly enforced that every agent is member of at most one group — and the second condition requires that the to-be-assigned deputy B is also member of the same group G. However, this only enforces that at the moment of deputy assignment, A and B are member of one common group, and in fact the high-level policy is violated as soon A or B change to another group while the deputy relation is in place. In fact a real system may be built like this and have the weakness that the meta policy is not checked

when people change groups. We thus see it as a strength of ASLan++ that such systems (with all their flaws) can be modeled and the problem be discovered by automated verification.

To formalize a system that realizes the deputy-in-same-group meta policy (no matter how), the easiest way is to actually allow in the model deputies outside the group, but to enforce the same-group constraints whenever access is granted on grounds of the deputy relation, i.e., refining our original `accessDeputy` rule:

```
accessDeputy(A,B,F,G) : A->canAccess(F) :-  A->deputyOf(B) & B->canAccess(F)
                                          & G->contains(A) & G->contains(B);
```

If there are other decision made based on the `deputyOf` relation, they would have to be refined similarly.

4 Channels

A very typical aspect of the systems we model in ASLan++ is that they are distributed and communicate over (initially) insecure channels that could be accessible to an intruder who may read, intercept, insert and modify messages. It is also common to secure the communication lines by protocols like TLS or IPSec and thereby obtain a virtual private network, i.e., as if the distributed components were directly connected by secure lines.

4.1 Abstraction Levels

ASLan++ is of course expressive enough to directly model protocols like TLS and IPSec, using the classical cryptographic primitives for encryption and digital signatures, but this is not really desirable: one should not model large systems monolithically and in all detail, but, whenever possible, distinguish different layers and components in a system. This approach entails to verify high-level applications that are run over secure channels independently of the low-level protocol that provides these channels. This gives also greater significance to the verification result: the application is then secure even when *exchanging* the low-level secure channel protocol. Vice-versa, the channel protocol should be verified independently of a concrete application, so that it can be used for other applications as well. There are first results for such *compositional reasoning* for channels [20,11].

ASLan++ supports an abstract notion of channels where we simply state that messages are transmitted under certain assumed security properties. We have already seen examples above:

```
?A *->* Actor: requestAccess(?F)
```

The stars mean that the respective side of the channel is protected. Protection on the receiver side means confidentiality: it can only be received by the intended receiver. Protection on the sender side means authentication: it is certain that the message indeed comes from the claimed sender. Authentication also includes that the intended recipient is part of what is being authenticated; so the receiver

can see whether this message was really intended for him (even though everybody can read it when confidentiality is not stipulated). This follows the definition of the *cryptographic channel model (CCM)* and the *ideal channel model (ICM)* presented in [20]. There is an alternative notation supporting the *abstract channel model (ACM)* of [2]; we leave this out here for lack of space and also because the integration into ASLan++ is not finished at the state of this writing. Much more detail on the three channel models may be found e.g., in [4, §3.8].

The channels we have in our little example only ensure confidentiality and authenticity/integrity, they do not incorporate other properties such as recentness (which can be achieved using the `->>`), disambiguation of different channels between the same principals (which can be achieved by including distinguishing channel/session identifiers in the messages) or the ordering of messages (which can be achieved by including sequence numbers). For details on these aspects of channel modeling we refer to [4, §2.9].

4.2 Client Authentication

We do want to illustrate however one very common situation in modeling channels: one side may not be authenticated. The most typical example is TLS where usually the server possesses a certificate, but the client does not. One may model this situation by declaring transmissions from client to server as being only confidential (but not authentic) and transmissions from server to client as only authentic (but not confidential). However, TLS with an unauthenticated client provides actually more security guarantees, namely sender and receiver invariance for the client: even though the client's request is not authenticated, the response from the server is sure to go only to that client who posed the request, and subsequent requests can be associated to the same client. [20] suggests regarding this as a secure channel except that the client is not authenticated by its real name but by a self-chosen pseudonym. We denote this such a channel in ASLan++ as `[Client]_[Pseudonym] *->* Server`.

Such unilaterally authenticated channels are of high relevance in practice for many applications, such as transmitting authentication information of the client like passwords or cookies to the server. In this case, the server can definitely link the pseudonym to the client's real name. If we wish to just abstract from the TLS channel, but not from the authentication mechanism that is run over the channel, then the pseudonymous channel notation of ASLan++ gives us the possibility to do so. Let us consider for that reason a refinement of our previous example. Before, we used a secure channel between a Manager M and the server S. Let us now model that M has a TLS channel to S where M is not authenticated. As a first step, the manager would send a login with his name and password. The password we model as symmetric key which is a function of M and S:

```
nonpublic noninvertible password(agent,agent): symmetric_key;
```

Here, `nonpublic` means that no agent itself can apply the password function, one can only initially know passwords or learn them during a message transmission. Similarly `noninvertible` means that one cannot obtain the agent names from

a given password. We ignore here bad passwords, but we explicitly allow the intruder to have its own password with S, namely `password(i,S)`, which is initially known to the intruder.

We model that when a manager logs in to the server (here the `Actor`) over the pseudonymous channel `[?M]_[?MP] *->* Actor`, the server creates a cookie for that manager, sends it back on the pseudonymous channel, and stores the cookie, along with the manager's identity, in a cookie database:

```
on([?M]_[?MP] *->* Actor: login(?M,password(?M,Actor)) & ?M->isManager): {
  Cookie := fresh();
  cookies(Actor)->contains((M,Cookie));
  Actor *->* [M]_[MP]: Cookie;
}
```

Note that in all transactions involving a manager we write `?M` and `?MP` because the identity of M and her pseudonym `?MP` are learned by the server at this point. This allows for modeling multiple managers. When a manager connects, stating its own name M, the server requires an abstract login message that consists of the user name M and password. After these have been verified, it makes sense to check if M is indeed a manager, which we describe by the predicate `isManager`. With the line `Actor *->* [M]_[MP]: Cookie;` we ensure that the cookie goes to exactly the person who sent the login (the owner of the pseudonym MP which is – hopefully – the manager). Note that this allows us to faithfully model also the situation where an intruder has found out the password of a manager: in this case he can now obtain also such a cookie (and use this cookie for subsequent impersonation of the manager).

The cookie database is also worth discussing in more detail: we declare

```
nonpublic cookies(agent): agent*cookie set;
```

i.e., similar to the passwords, it is a function parameterized over agent names — in this case the owner of the database. The cookie database simply consists of a set of pairs of agent names and cookies.

We can now re-formulate the `changeGroup` action of the manager to run over a pseudonymous channel, using a previously obtained cookie for authentication:

```
on([?M]_[?MP] *->* Actor: ?Cookie.(?A->changeGroup(?G1,?G2))
    & cookies(Actor)->contains((?M,?Cookie)) & ?G1->contains(?A)): {
  retract(G1->contains(A));
  G2->contains(A);

}
```

Here the manager is authenticated by the cookie, which is looked up, in conjunction with her name stored in the variable M, in the server's cookie database before granting the transaction. Again this is a faithful model of the real situation: we send a cookie over a TLS channel where the sender is not authenticated — possibly even the pseudonym MP is not the same because a new TLS session had been opened meanwhile. If for some reason the intruder has obtained such a cookie, he can use it to impersonate the manager in such transactions.

This example illustrates how the channel notation can be used to model different levels of granularity of our models: we can either completely abstract from authentication mechanisms and right away use a secure channel, as we did

first, or we can just abstract from TLS but model a credential-based approach like the above password/cookie mechanism. The abstraction has the advantage that we fade out model details and make thus the specification easier to read and work with, while a more detailed specification may allow to model more aspects such as leaking passwords or cookies. Note that again such an intermediate layer could also be addressed with compositional reasoning, i.e., specifying just the credential-based system without concrete applications like `changeGroup` and authentic transmission as a goal.

5 Security Goals

ASLan++ has been geared as a high-level input language for model checking security aspects of distributed systems, and it is therefore crucial to offer a convenient, clear, and expressive way to formalize the desired security properties. The most general way to describe a security property in ASLan++ is to use a first-order temporal-logic formula, defining a set of traces G that satisfy the security properties. An *attack* is then any trace that the system can show and that is not contained in G. The logic that we use is an extension of *LTL* (linear temporal logic); for brevity we refer to it simply as LTL. The propositional basis are the ASLan++ facts, and we allow all the standard temporal operators from LTL and first-order quantification. The currently available tools however support only fragments of this logic. First, all tools currently only support outermost universal quantification. Second, OFMC and CL-AtSe support only safety properties, i.e. such that every attack manifests itself in a finite trace, while SATMC also supports liveness properties. We do not support any properties that involve multiple traces such as non-interference goals.

To make the specification of simple and common goals as convenient as possible, especially to user without a background in formal logic, we provide several ways to specify goals. In particular we can formulate goals within an entity, allowing to refer to all variables defined in this scope.

Invariants. Goals that should hold during the whole life of an entity instance are stated in the `goals` section of the entity declaration. Properties expected to hold globally during the overall system execution should be given in the outermost entity.

For invariants, basically any LTL formula can be given. As an example, consider the meta policy "one cannot have a deputy outside one's own group" mentioned in 3.5:

```
goals
 deputy_in_group: forall A B. [](B->deputyOf(A) =>
                                 (exists G. G->contains(A) & G->contains(B)));
```

where "`[]`" is the "globally" LTL operator. However this is outside the supported fragment of all tools (due to the existential quantifier). Taking advantage of the fact that in our model each employee is in exactly one group, which could be specified and checked as a further invariant, we can re-phrase the formula as

```
forall A B G. [](B->deputyOf(A) => (G->contains(A) => G->contains(B)))
```

As mentioned before, this goal is violated.

Assertions. An assertion is very similar to an invariant, except that it is inserted as a statement in the body of an entity and is expected to hold only at the given point of execution of the current entity instance.

In our example, we can express the expectation that an employee is allowed to access a certain file `F` using a very simple LTL formula:

```
assert can_access_file: Decision = grantedAccess(F);
```

Channel Goals. ASLan++ offers special support for conveniently specifying the usual communication goals like confidentiality, authentication, and the like, called *channel goals*. In our example, in order to state that a manager authenticates to the server on her cookie when sending a `changeGroup` command, we write

```
manager_auth:(_) M *-> S
```

In analogy to the syntax of message transmission, `M` denotes the sender and `S` denotes the receiver of the transmission to which the goal refers. In between them is a symbol for the kind of channel property, in this case "`*->`" indicating sender authenticity. The goal name `manager_auth` is augmented with a parameter placeholder "`(_)`" to indicate that in sub-entities the goal name appears again, in the form of a goal label, with a message term as its argument. Here, the message term is `Cookie`.

This channel goal is stated, as usual, at the level of the `Session` entity and pertains to those message transmissions in which the goal name `manager_auth` re-appears. In our example, we write in the `Manager` entity:

```
[Actor]*->* S: manager_auth:(Cookie).
               (e1->changeGroup(g1,g2));
```

and in the `Server` entity:

```
[?M]_[?MP] *->* Actor: manager_auth:(?Cookie).
                       (?A->changeGroup(?G1,?G2))
```

The operational semantics of this goal is that whenever the server receives `?Cookie.(?A->changeGroup(?G1,?G2))` from any manager `?M`, the agent denoted by `M` must have sent to the server the `changeGroup` command with the same cookie value (as long as `M` is not the intruder legitimately playing the manager's role).

It is important to note that `M`'s value is determined dynamically here, depending on the cookie just received. The side condition

```
cookies(Actor)->contains((?M,?Cookie))   % actually learns M here!
```

models that the server looks up the name `M` in its database: it is the name that was stored along with the cookie when the manager logged in. So the manager to be authenticated is not determined statically by the initial value of the parameter `M` of the `Server`.

Secrecy Goals. Very similarly to the channel goal just described, we state

```
shared_secret:(_) {M,S}
```

in the `Session` entity. Its interpretation is that the values annotated by the respective goal labels in the sub-entities must be known only to the agents `M` and `S`. In this case, the confidential value is the password that the manager uses for logging in to the server. Therefore, we write in the `Manager` entity:

```
[Actor]*->* S: login(Actor,shared_secret:(password(Actor,S)));
```

and in the `Server` entity:

```
on([?M]_[?MP] *->* Actor: login(?M,  % actually learns M here!
                          shared_secret:(password(?M,Actor)))
```

The operational semantics is that after the manager or the server has processed the value given as argument of the goal label `shared_secret:(...)`, this value must never show up in the knowledge of the intruder (as long as the intruder does not legitimately play the role described by any of the two entities).

Confidential transmission of a value between two parties can also be stated as a channel goal, but the secrecy goal is more general: it may be used to state that the value is shared between more than two parties, and the confidentiality is meant to be persistent (unless the secrecy goal is retracted or modified dynamically).

6 Conclusion

We have illustrated by means of an example how the major security-relevant features of modern service-oriented architectures can be specified in ASLan++, in particular how to formulate non-linear and inter-dependent workflows, as well as policies and related goals. We have shown how to selectively abstract from communication aspects using secure channels, as well as pseudonymous channels and password- or cookie-based authentication mechanisms. The specifications are clear and readable for web service designers, and at the same time are on a reasonable abstraction level to be feasible for automated verification tools such as those of AVANTSSAR. For instance, the violation of the `deputy_in_group` goal in our example is found by the CL-AtSe back-end in less than a second.

AVANTSSAR Case Studies and Tool Availability. While the running example has been deliberately kept small and simple for presentation, this paper reflects also our experiences in the AVANTSSAR project with real-world case studies [3] from the areas of e-Government, e-Health, and e-Business. There, ASLan++ similarly allows us to have well-structured, easy-to-read descriptions of complex systems that can be effectively analyzed with the automated verification tools of the AVANTSSAR platform within reasonable time (usually much less than 1 hour CPU time). Both the case studies and the AVANTSSAR Tool, including a convenient web interface, are available at `www.avantssar.eu`.

Related Work. There are a number of specification languages that have similar or overlapping aims. The closest ones are the high-level protocol specification language HLPSL [10] and the low-level language IF of the predecessor project AVISPA [5]. In fact, experience with these languages had much influence on the ASLan++ design. The most crucial extensions w.r.t. HLPSL and IF are the integration of Horn clauses and the notion of channels. Moreover, ASLan++ is closer to a programming language than the more logic-oriented HLPSL and the more low-level description of transition rules of IF. ASLan++ is automatically translated to the more low-level language ASLan, which is an appropriate extension of IF and serves as the input language of the model-checking tools.

In the area of policy specification languages, we are closest to SecPAL [6] and DKAL [18] with their Horn clause specification style. their relation with a transition system. Another, conceptually quite different approach is KLAIM [14], which allows for specifying mobile processes where access control is formalized using a capability-based type system. Despite the differences, the combination of a policy aspect and a dynamic distributed system bears similar ideas and we plan to investigate as part of future work whether the concepts of the two languages could be connected.

A language focussing on the vertical architecture especially in web services is Capito [17], this is however again built on relatively simple authentication protocols and is not related to the required compositionality results such as [20,11].

One of the pioneering verification frameworks for web services is the Tula-Fale project [7], which in particular supports a convenient way to deal with the details of the message formats such as SOAP. It is based on the verification tool ProVerif [8] using abstraction methods, which represent the entire protocol as a set of Horn clauses. A limitation of this approach is the monotonicity the abstraction, which forbids for instance to model revocation of access rights. One of our works aims to overcome this limitation while preserving the advantages of abstract interpretation, namely the set-based abstraction approach [19]. It is part of our future work to build a bridge from ASLan++ to that framework. This bridge will consist not only in a translator from ASLan++ to a suitable input language, but also of a mechanism to choose and refine appropriate abstractions.

Acknowledgments

The work presented in this paper was supported by the FP7-ICT-2007-1 Project no. 216471, "AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures". We thank the anonymous reviewers for their helpful comments.

References

1. A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron.

The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005. http://dx.doi.org/10.1007/11513988_27.
2. A. Armando, R. Carbone, and L. Compagna. LTL Model Checking for Security Protocols. In *Journal of Applied Non-Classical Logics, special issue on Logic and Information Security*, pages 403–429. Hermes Lavoisier, 2009.
3. AVANTSSAR. Deliverable 5.3: AVANTSSAR Library of validated problem cases. www.avantssar.eu, 2010.
4. AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial. www.avantssar.eu, 2011.
5. AVISPA Project. www.avispa-project.org.
6. M. Y. Becker, C. Fournet, and A. D. Gordon. Security Policy Assertion Language (SecPAL). research.microsoft.com/en-us/projects/SecPAL/.
7. K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. Tulafale: A security tool for web services. In *Proc. 2nd FMCO*, LNCS 3188, pages 197–222. Springer, 2003.
8. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of CSFW'01*, pages 82–96. IEEE Computer Society Press, 2001.
9. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
10. Y. Chevalier, L. Compagna, J. Cuéllar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Automated Software Engineering. Proc. SAPS'04 Workshop*, pages 193–205. Austrian Computer Society, 2004.
11. S. Ciobâca and V. Cortier. Protocol composition for arbitrary primitives. In *Proceedings of CSF*, pages 322–336, 2010.
12. H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. Technical Report LSV-03-3, Laboratoire Specification and Verification, ENS de Cachan, France, 2003.
13. C. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *CAV 2008*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.
14. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a kernel language for agents interaction and mobility. *IEEE TSE*, 24(5):315–330, 1998.
15. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.
16. S. Escobar, C. Meadows, and J. Meseguer. Maude-npa: Cryptographic protocol analysis modulo equational properties. In *FOSAD*, pages 1–50, 2007.
17. H. Gao, F. Nielson, and H. R. Nielson. Protocol stacks for services. In *Proc. of the Workshop on Foundations of Computer Security (FCS)*, July 2009.
18. Y. Gurevich and I. Neeman. Distributed-Knowledge Authorization Language (DKAL). research.microsoft.com/~gurevich/DKAL.htm.
19. S. Mödersheim. Abstraction by Set-Membership—Verifying Security Protocols and Web Services with Databases. In *Proceedings of 17th CCS*. ACM Press, 2010.
20. S. Mödersheim and L. Viganò. The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols. In *Fosad 2007-2008-2009*, LNCS 5705, pages 166–194. Springer-Verlag, 2009.
21. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *CSFW*, pages 174–. IEEE Computer Society, 2001.

ASLan++ specification example

```
specification example
channel_model CCM

entity Environment {

  types
    file < text;
  % a group is an agent set
    cookie < text;

  symbols
    login (agent,symmetric_key): message;
    changeGroup (agent,agent set,agent set): message;
    assignDeputy(agent): message;
    requestAccess(file): message;
    grantedAccess(file): message;
     deniedAccess(file): message;

    nonpublic noninvertible password(agent,agent): symmetric_key;
    nonpublic               cookies(agent): (agent * cookie) set;
    % used by the Server to store cookies for managers

    centralServer: agent;
    isManager(agent         ): fact;
    canAccess(agent    ,file): fact;
    isOwner  (agent set,file): fact;
    deputyOf (agent   ,agent): fact;

  clauses
    accessDirect(A,G,F): A->canAccess(F) :- G->isOwner(F)  & G->contains(A);
    accessDeputy(A,B,F): A->canAccess(F) :- A->deputyOf(B) & B->canAccess(F);

  symbols % for the concrete access examples
    m1: agent;
    e1, e2, e3: agent;
    g1, g2: agent set;
    f1, f2: file;

  entity Session (M, S: agent) {

    entity Server(M, Actor: agent) {
  % Exercise for the reader: how to formulate this for a decentralized system?
  % Hint: introduce either an additional argument (representing the P.o.V.) to
  % all policy judgements, or a modality like "Server->knows(A->canAccess(F))".
      symbols
        MP: public_key; % pseudonym of a manager
        A, B: agent;
        G, G1, G2: agent set;
        F: file;
        Cookie: cookie;
      body {
        while(true) {
          select {
            on([?M]_[?MP] *->* Actor: login(?M,  % actually learns M here!
                                            shared_secret:(password(?M,Actor)))
               & ?M->isManager): {
              Cookie := fresh();
              cookies(Actor)->contains((M,Cookie));
              Actor *->* [M]_[MP]: Cookie;
            }
            on([?M]_[?MP] *->* Actor: manager_auth:(?Cookie).(?A->changeGroup(?G1,?G2))
               & cookies(Actor)->contains((?M,?Cookie))   % actually learns M here!
               & ?G1->contains(?A)): {
              retract(G1->contains(A));
              G2->contains(A);
            }
```

```
            on(?A *->* Actor: assignDeputy(?B) & ?G->contains(?A) & ?G->contains(?B)): {
              B->deputyOf(A);
            }
            on(?A *->* Actor: requestAccess(?F)): {
              if(A->canAccess(F))
                Actor *->* A: grantedAccess(F);
              else
                Actor *->* A:  deniedAccess(F);
            }
          }
        }
      }
      goals
        deputy_in_group: forall A B G. [](B->deputyOf(A) =>
                                          (G->contains(A) => G->contains(B)));
    }
    entity Manager(Actor, S: agent) {
      symbols
        Cookie: cookie;
      body {
       [Actor]*->* S    : login(Actor,shared_secret:(password(Actor,S)));
        S    *->*[Actor]: ?Cookie;
       [Actor]*->* S    : manager_auth:(Cookie).
                          (e1->changeGroup(g1,g2));
      }
    }
    body { % of Session
      iknows(password(i,S)); % intruder knows its own password
      new Server (M,S);
      new Manager(M,S);
    }
    goals
      shared_secret:(_) {M,S};
      manager_auth :(_) M *-> S;
  }
  entity Employee(Actor, S: agent) {
    symbols
      F: file;
      G: agent set;
      Decision: message;
    body {
      if(Actor=e1)
        Actor *->* S: assignDeputy(e2);
      % results in a meta policy violation if "e1->changeGroup(g1,g2)" happens later!

      % get any file currently owned by this employee
      if(?G->contains(Actor) & ?G->isOwner(?F)) {
        Actor *->* S   : requestAccess(F);
        % before the decision is received, access rights could have changed...
        S    *->* Actor: ?Decision;
        assert can_access_file: Decision = grantedAccess(F);
      }
    }
  }
  body { % of Environment
    % for the concrete access examples:
    m1->isManager;
    g1->contains(e1); g1->contains(e2);
    g2->contains(e3);
    g1->isOwner(f1);
    g2->isOwner(f2);

    any M. Session(M,centralServer);  % M may be dishonest!
    new  Employee(e1,centralServer);
    new  Employee(e2,centralServer);
  % new  Employee(e3,centralServer);
  }
}
```